RSRE

MEMORANDUM No. 4249

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE NEW PROGRAMMING

Authors: C S L Phillips, B D Bramson

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4249

CONDITIONS OF RELEASE

0048217 BR-111061

********** U

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4249

THE NEW PROGRAMMING

C S E Phillips
Edited by
B D Bramson

May, 1989

Abstract

The late C S E Phillips's unfinished paper is presented. The sequential mode of thinking that underpins conventional programming is cited as being unnecessary, over-complicated and impractical for all but the most trivial of problems. In *New Programming*, the flow of data is the only factor of importance. A *New Computer* is a non-sequential machine whose elemental processors are not required to execute their processes in an ordered sequence. Such a machine is easier to design and easier to prove correct than its sequential counterpart.

Contents

1 EDITOR'S PREFACE 1

2 INTRODUCTION 1

3 A GENERAL MODEL OF PROCESSING SYSTEMS 4

4 NETWORKS OF STORES AND PROCESSORS 6

5 THE LANGUAGE OF NEW PROGRAMMING 9

6 PROVING NEW PROGRAMS CORRECT 11

7 EXAMPLES 12
- 7.1 FIVE PROBLEMS . . . 12
- 7.2 TOP LEVEL DESCRIPTIONS . . . 12
- 7.3 DECOMPOSITION . . . 13
- 7.4 QUADRATIC EQUATIONS . . . 14
- 7.5 OFF-LINE IMPLEMENTATION . . . 15
- 7.6 MINIMISING STORAGE . . . 16

8 OLD PROGRAMS VIEWED AS DATA PROCESSING NETWORKS 17

9 TRANSLATING OLD INTO NEW PROGRAMS 18

10 ORGANISATIONAL BENEFITS 20

11 EDITOR'S POSTSCRIPT 20

1 EDITOR'S PREFACE

Shortly before he died in 1980, Dr C S E (Bob) Phillips was investigating an approach to the design of computer programs targeted at the so-called "software crisis". In a sense, this was the bringing together of two of many themes in his versatile research career. The first involved a graphical scheme for representing real-time systems, both hardware and software, in which different processes shared data [1]. Diagrams that obeyed the rules of the scheme soon became known as *Phillips diagrams* and these had a profound influence on the MASCOT design philosophy that evolved later [2]. Quite separately, Phillips pioneered techniques for analysing the logical structures of sequential programs [3]. These aided comprehension; were seen as a first step towards software verification; and sowed the seeds for what eventually became the Malvern Program Analysis Suite (MALPAS).

Very soon, Bob Phillips realised that producing programs capable of being proved correct was a tricky matter. The problem lay, he claimed, with the very nature of sequential programs, with the use of selective modules, loops and worst of all shared storage. With decreasing hardware costs, alternative architectures had to be the answer. His paper, which appears to be a first draft, was given to me by the author in 1980. I must therefore apologise for having been so late in bringing it to the attention of a wider audience.

The style is informal but the approach is directly relevant to modern developments in formal specification, design and verification. Sadly, one page of the original typescript was missing along with the entire bibliography cited by the text. I have therefore added some obvious references. I have also made several changes of an editorial nature together with some footnotes that reflect my interpretation of the text. It has been my intention, however, to retain the original flavour of the paper wherever possible.

2 INTRODUCTION

The cost of computer hardware has fallen dramatically over the past 30 years, causing an expansion in the demand for programmers and for programs of ever increasing complexity. On the other hand, the costs of programming have fallen comparatively little over this period. Unfortunately, the problems of writing, debugging, testing, understanding, modifying, analysing, and proving programs increase more than proportionately with size; and these problems are exacerbated when teams of programmers are needed. This situation is sometimes described as the "software crisis".

It is generally recognised that the methods of program production, adequate for simple programs written in a few days by a single programmer for his own purposes, are unsuitable for larger scale organised production. By analogy with engineering and other established disciplines, various managerial and design techniques have been advocated. These are based on the modularisation, structuring and documentation of programs. Although the motives are correct, the view taken in this paper is that the proposed techniques are palliatives. The origin of the problem is much more fundamental, namely with computer programming itself. What is required is a new kind of programming that follows the style of thinking found in other, established disciplines.

The fault lies not so much with conventional programming but with the *sequential* mode of thinking on which it depends. The idea that a system should be constructed on the basis that its overall behaviour is to depend on the order in which its various parts

operate is used in few other disciplines [1]. The sequential algorithm, being inherently more complicated than an *order independent* "algorithm", is usually reserved as a way of explaining *how* something works or is to be used and not, if at all possible, as a recipe for its construction. Of course, it is easy to understand why sequential thinking dominates in computing since it mirrors the way conventional computers actually work. Indeed, the word "algorithm" usually implies sequentiality.

The starting point for this paper is therefore the observation that conventional programming is an *unnecessary, over-complicated* and *impractical* technique for constructing data processing systems of any kind save for the most trivial. Of course, the meaning of "trivial" depends on the level of language employed: for example, a program that is simple and easy to understand when expressed in Algol 68 might be non-trivial if written in machine code and even less trivial if written in microcode. Roughly, if a program can be written and developed by a single programmer in a few days and can easily be understood by another programmer, then the program is trivial; otherwise the program is too complicated and should not have been written.

New Programming is based on a non-sequential mode of thinking similar to that used in other disciplines. This in turn presupposes a non-sequential computer in which the flow of data is the only factor of importance; so that, to obtain a correct result, the elemental processors out of which it is constructed are not required to carry out their processes in an ordered sequence. Fortunately the "New Computer" need not be built of hardware, but can be constructed from software. The software consists of a set of conventional programs, designed to be as simple as one wishes, that intercommunicate via data "stores". For normal off-line work an entirely distinct, sequencing or controlling program is used to "run" the individual programs in a serial order unrelated to the state of the data in the system. Further, we may continue to use existing computers, languages and operating systems, changing nothing but our concepts.

For reasons of brevity, conventional, sequential programs will henceforth be referred to as *old programs* in this paper [2].

New Programming turns out to be extremely easy compared with old programming, at least as far as its mechanics are concerned, though to understand it requires a complete change of outlook. Almost every concept in old programming is inapplicable. First, we must understand the workings of a conceptual, non-sequential, "data flow" machine. Unlike a conventional computer, this is not a general purpose machine for interpreting programs: rather it is a special purpose machine *defined by* the New Program. Each particular problem is solved by *specifying* and *designing* a network of data processors and stores in a manner analogous to the design of any form of processing system. The mechanical interconnection rules are extremely simple. The difficulties of New Programming are mainly the difficulties of design, in other words of specifying the network. Thus, knowledge based on previous experience is essential.

By contrast, old programming is an inventive, meccano-like technique for constructing by evolution a network *of fixed abstract* parts. However, the interconnection rules in old programming are far more complicated than in New Programming. They involve processor sequence as well as data flow together with the concepts of *variable* processors and stores. The resulting network is virtually impossible to comprehend and its correctness may be

[1] eg although the different stages of a radio set process data in sequence, each stage runs in parallel.
[2] In fact the author retained "conventional" throughout.

impossible to prove in practice. Thus, unlike a New Program, an old program can be developed only by trial and error methods. Nevertheless, old programming, at least in its purest form, has the advantage of being a learning process. Thus, knowledge based on previous experience is unnecessary. Viewed in this light, it can be seen that the "software crisis" is a myth. Instead, there exists a "programmer crisis" in the sense that "pure" programmers must choose between:

1. "trivialising" their functions and

2. gaining the specialised knowledge needed to design data processing systems by becoming systems designers who write their designs as New Programs.

With hindsight, it is clear that New Programming is not really new, but a formalisation of previous *ad hoc* software design techniques. Its origins lie in a practical method developed some years ago for the production of software for an on-line, real-time, computer controlled radar [1]. This resulted in a network of small, apparently autonomous, concurrent programs that communicated with each other via data "areas" (stores and buffers of stores). Subsequently, this simple and apparently self evident technique was applied more widely; but it soon became clear that the very concept of data flow, that systems designers and non-programmers, had no difficulty in understanding was found by programmers to be hard to relate to programming. Indeed the author and others made efforts to extend the network symbols so as to include the far more complicated concepts of ordinary programming. Such diagrammatic extensions to "program networks", as they became called, were never wholly satisfactory for reasons that are now obvious, for if successful they would have constituted a method for proving correctness. (A practical way of proving the correctness of an old program is to *transform* it into an equivalent data flow network form.) An important complicating factor, now seen to be essentially irrelevant, was a confusion between on-line and off-line systems and between parallelism and non-sequentiality. This was caused by the practical need to implement a data flow network on a single computer.

Thus, there was conceptual confusion on the one hand and the difficulty of enforcing an unfamiliar design method on the other. A considerable advance occurred with the advent of MASCOT [2, 4] and its associated method of constructing systems based on *enforceable* data flow networks. MASCOT has now been applied successfully to several systems so that New Programming under another name is now well-established, at least for on-line systems.

Data flow design techniques have also been established for off-line applications. Quite recently [3], several papers have appeared that reveal essentially the same data flow philosophy for non-numeric applications, for general purpose computing and for the design of computer networks. Doubtless, similar ideas have occurred to a number of practising programmers. However, the design techniques described in these papers all refer to special implementations which, though of practical value, limits their immediate applicability and obscures the underlying theoretical concepts. Evidence that special operating systems are unnecessary (though convenient) is provided by the many programmers who have invented in effect a serial form of the data flow network by designing programs as series of "passes". The "top" level of a "structured" program is also in effect a serial data flow system, but

[3] ie 1980

the subsequent expansion of selective modules is based on conventional sequential programming ideas. Constantine's data flow graphs, described as "bubble" charts by Brown, reflect a data flow philosophy even though the important data "areas" are not included. Consequently, the significance of the charts does not appear to have been properly understood.

3 A GENERAL MODEL OF PROCESSING SYSTEMS

A data processing system, consisting of hardware, software or a set of people in a bureaucracy, may be regarded as a discrete product processing system whose products are informational. Such a system inputs a number of different objects of one kind and outputs a number of different objects of another kind. Among many examples, we can think of a country importing and exporting goods, a factory producing a range of articles from various parts, transport systems and financial systems. The functional transformation between inputs and outputs is called a *process* and is performed by a *processor*. Of course, some processors merely transfer products from one place to another (eg transport systems, shipping).

A processing system has only two parts:

1. *processors* (eg the country, factory or transporting device) and
2. *product buffer stores* (eg warehouses) where the products are kept when not being processed.

Like any system, a processing system can be viewed either as subdivided into sub-systems or as their aggregation. For example, a country with its customs warehouses may be regarded as the sum of its separate factories and warehouses. Similarly, the world processing system may be viewed as a "closed" system, with no imports and exports, consisting of all internal countries, customs warehouses and shipping.

We may therefore model the structure of a product processing system as a network of processors and product buffer stores. We can express such a structure either as a set of functional equations or, more conveniently, as a diagram in which processors and product stores have different symbols and where *directed arcs* represent the directional flows of products from stores to processors and processors to stores.

A product store is simultaneously a *source* for some processors and a *sink* for others. If the system flow is continuous and if the processors are autonomous, so that the rates of flow are uncontrolled, the number of items of the same type in a buffer will vary. Thus, the buffers will be empty on some occasions and full at others. Further, a product buffer store can contain more products than are immediately required as inputs to processors.

Let us imagine that a buffer store is subdivided into separate *stores* each containing an individual product. Thus, a buffer store is simply a set of stores containing similar products. Each buffer, and therefore each of its stores, can be identified by the name [4] of the product it contains. Consider, for example, a buffer store named "car parts" divided into a set of stores also called "car parts". (We may suppose that "car parts" is used by a processor named "car factory".) The "car part" buffer store would consist of a set of similar car parts stored at different times.

[4] In this context, the reader may prefer the term TYPE.

A buffer thus contains a *history* of equivalent products that could be used by processors in any order. It follows that we can distinguish

1. the structural design of a processing network comprising named processors and stores, and

2. the problems of flow control that necessitate buffering.

Since we are interested primarily in structural design we shall ignore buffering for the present and refer simply to *stores* which contain products.

When a processing system is modelled by a network it is usually clear which are the processors and which products are input and output, but the naming of product stores is somewhat arbitrary. Naming implies the wish to think of something as a whole even though it may be divisible. On the other hand, a named product store could contain a product part of which was provided by one processor and part by another. In network terms, both the in-degrees and out-degrees of store *nodes* (the numbers of incoming and outgoing arcs) would be unrestricted. In such a system the names of stores would be *composite*, in the sense of referring to arbitrary groups of sub-products produced and used by processors, and these would have more to do with naming and designing stores than with the processing *system*. If group names were associated with stores rather than with products, we could imagine stores becoming re-used at different times for different products. Such matters would be important when designing a system to minimise overall storage requirements but they greatly complicate the processing system.

To avoid these complications, we shall distinguish sharply between the design of a system and its implementation, there being no distinction in the former between the name of a store and the name of the product it contains. The minimisation of storage requirements shall be regarded as a separate question (like flow control) to be dealt with at a later stage. It follows that we must identify the names of stores either with the names of the products of processors or with the items used by processors or both. For example, we could split "car parts" into "car engines" and "non-engine car parts" on the grounds that a factory making car engines exists or that a factory exists which uses engines only or for both reasons. In other words, we could associate stores with producers, consumers or both.

We choose not to associate stores with consumers, since this would lead to highly complex functional equations. Instead, we shall demand that each product emanates from one processor only, so that the in-degree of each store node will be unity. On the other hand, the out-degree is left optional on the grounds that "use" can mean "input"; thus, stores are always "emptied" by processors whether or not all the parts of a product are needed. It follows that in expanding or contracting the design of a processing system, we expand or contract stores in terms of their internal sub-components without concerning ourselves with these sub-components at any fixed level of design. Of course, the processors themselves "understand" the structure of the product store but from the viewpoint of the network both processors and stores are "black boxes".

In certain processing systems the products are not discrete. Examples include fluids in chemical processing plants and signals in communications systems. However, by quantising the fluids or signal amplitudes in time, the systems that process them can also be regarded as discrete processing systems. For communications systems the product is no longer physical but informational and the concept of an *empty* store is meaningless (eg signals in the aerial of a radio or television set). In such cases certain *characteristics* of the product

are processed rather than the product itself. For example, in communications we may be interested in the numerical values of the signal modulation expressed as a function of time over the quantum interval. Such "data" *replaces* the original data even though many receivers can input the original data simultaneously.

Thus our data processing model must be based upon individual stores associated with sets of values that are *read* by processors *non-destructively* and *written* by processors *destructively*. A data *buffer* on the other hand is read destructively and written non-destructively since in this case we are removing or adding stored products. This viewpoint is entirely applicable to computers for all internal storage (core, disc, magnetic tape). On the other hand, unlike conventional usage where composite names are used, the terms store and buffer here refer to names of data occupying distinct storage locations (though the precise locations need not be specified).

In a computer-based data processing system we are often concerned with a single set of inputs and outputs, the flow being *discontinuous*. We shall regard such an "off-line" system as a special case of a data processing system for which buffers are unnecessary and therefore non-existent. The *design* of such a system is exactly the same as for a continuous flow system, but the *implementation* will be different in that flow control problems do not exist. Also there will be greater opportunities for subsequent minimisation of total storage.

4 NETWORKS OF STORES AND PROCESSORS

A *New Program* describes a data processing system comprising a fixed set of data *stores* containing *values* (corresponding to mathematical variables) and a fixed set of *processors* that perform invariant data transformations expressed by mathematical functions. Each processor also defines a fixed subset of stores from which it *inputs* values and another fixed subset of stores to which it *outputs* values. Different processors may perform the *same* function. Stores are finite but conceptually unbounded in size, but stores are *not* buffers. Since most of the input stores of each processor are the output stores of other processors, such a system can be described either by a set of functional equations or by a *network* [5] of two types of *node* connected by directed arcs. Figure 1 shows a simple example of an *acyclic* network (ie without loops) composed of four processors and nine stores. The stores are drawn as *rectangles* with their names inserted where we imagine the values to reside. The processors, regarded as "black boxes", are drawn as *circles* with the functions named inside. The connections between stores and processors are drawn as directed arcs through which we imagine values to flow. Copies of values flow *from* stores and new values flow *into* stores.

In such a data processing network some of the stores are external. Stores with no incoming arcs are described as the *external inputs* to the system (eg S_1 and S_2) while those with no outgoing arcs are the *external outputs* (eg S_7 and S_8). All other stores are internal to the system. The in-degree (number of incoming arcs) of an internal store is normally *unity*. When values are placed in a store by a processor, the new value replaces the old value and the variable is said to be *defined*. When a processor inputs a value from a store, the value in the store remains unchanged and the variable is said to be *referenced*.

In physical terms each processor must initially be triggered (switched on) by some

[5] ie directed graph

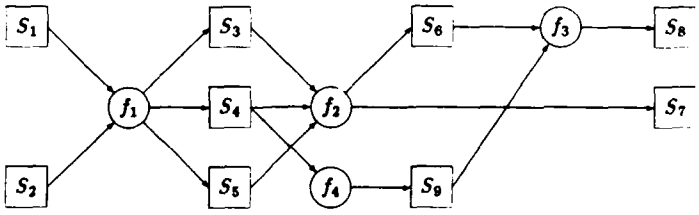

Figure 1: A Data Processing Network containing stores □ and processors ○.

means *external* to the system; processors cannot trigger each other. Each processor then carries out its process: it inputs values, performs its function, outputs values and terminates. A process thus occupies a finite time, called its *period*, which may depend on the values of input data [6].

Suppose that new values are placed in the external input stores (eg S_1 and S_2) by means *outside* the system and that the processors (eg f_1) that input these stores are triggered. Then these input processors will eventually change the values of their output stores in a time depending on their periods. By triggering each processor in turn as its inputs become redefined (eg following the sequence f_1, f_2, f_4, f_3), a flow of redefined values passes through the system finally redefining the external outputs. The time taken to do this is called the *period* of the system. A system in which the external input variables are defined on one occasion only is termed a *discontinuous data flow system* or off-line system. When a serial set of processes are synchronised to the flow of redefinitions the system is said to employ *synchronous serial processing*.

An off-line system can be implemented on a single, conventional computer. The individual processors are replaced by old programs that are "run" in sequence under the control of an entirely separate, conventional operating system. It is an important principle of New Programming that this sequence shall not normally depend on the state of the system (ie the current values of the variables). Data flow *loops*, however, present an exceptional case.

If the same sequence of processes were continuously repeated, the same flow of redefinitions would take place repeatedly and the state of the system would never change. If, however, all the processes were repeated *continuously* but in *any other* sequence, various *invalid* redefinitions would initially flow throughout the system; but eventually, though more slowly than before, *valid* redefinitions would pass through the system and the correct final state would be achieved. The time taken to do this is again called the *period* of the system. Such a system is said to employ *asynchronous serial processing* and could be used for on-line systems where only a single computer is available and data flow rates are being optimised.

We turn now to *asynchronous parallel processing*. After being switched on, each processor repeats its process autonomously and unendingly *in parallel* with other processors. Examples may be found in networks of computers, in hardware digital processing, in "data flow computers" and in organisations of people. With asynchronous parallel processing as

[6] In a real-time system where processes are repeatedly performed, termination means the completion of a cycle.

with asynchronous serial processing the phenomenon of transient, invalid definitions will again be found but its duration will usually be shorter. The time taken to complete the first flow of *valid* definitions, the period of the system, will also be shorter at least for networks with parallel data flows. However, for serial data flow problems, where *valid* processing cannot be performed in parallel, asynchronous parallel processing would be somewhat longer than synchronous serial processing.

A more complicated method that synchronises processes to the flow of definitions involves transmitting flags, semaphores or tokens with the data, as with Petri nets. The period of such a *synchronous parallel processing* system is minimal, ignoring overheads required to set and test flags, since each processor can "idle" with a very short period while attending a redefinition. This method is used in computers with so-called "autonomous data transfers" (where the processor is hardware) and "interrupt programs". Flag synchronisation can also be used with serial processing to ensure that a process is not performed unless all its input data flags are set, but it is only strictly necessary for the handling of data flow loops in discontinuous data flow systems. However, it is also discussed in the literature on continuous data flow systems where the individual processors are rather misleadingly described as "co-operating" [7].

There is an important difference in principle between asynchronous and synchronous processing. In the former, the arrival of *valid* definitions in the external output stores must be estimated since these are preceded by *invalid* definitions. In the latter, the arrival is known by the fact that any changes in the values of external outputs must be valid changes. In practice, this difference is less important than is generally supposed, since few output devices have the ability to record or follow rapid fluctuations.

In an on-line system, external processors (human or equipment) provide a continuous *stream* of redefinitions of the input variables causing a stream of new valid values to pass through the system such that the values of the external output variables will *vary with time*. Such a system is described as a *continuous* data flow system. If the frequencies of redefinitions of the various external input variables, the input data rates of the system, are sufficiently low the values of the external output variables will be valid; but if these rates are increased progressively there comes a point when the periods of the various processes are too long for the production of valid outputs. At this point the system breaks down, there being so-called "real-time" problems. However, a continuous data flow system would recover as soon as the input rates were reduced, though some of the previous data would have been lost. Replacing the stores by *buffers* can ease the problem, but improvement is limited.

To summarise, there are three aspects to New Programming that can be treated independently. The first is the design of a network for correctness by writing a New Program to ensure a *correct* solution to a given problem. The second is the choice between various synchronised or unsynchronised forms of serial or parallel processing. The third is the means of implementation on actual physical facilities, including issues of economy by sharing physical storage space. Where real-time systems are concerned, there is the further important matter of *flow control* which again can be treated quite separately.

[7] There may be some confusion here between co-operating with each other and co-operating with the data.

5 THE LANGUAGE OF NEW PROGRAMMING

Each individual processor may be regarded as performing a data transformation between its input and output stores expressed by a *functional equation* of the form

$$(y_1, y_2, y_3, ...y_n) = f(x_1, x_2, x_3, ...x_m). \tag{1}$$

The dependent variables $y_1, y_2, ...y_n$ correspond to the output stores; the independent variables $x_1, x_2, ...x_m$ correspond to the input stores; and the function f is performed by the processor. Replacing the independent variables by constants, or fixed values, corresponds to defining their values. It is for this reason that variables in New Programming are mathematical variables rather than program variables [8] and functions are mathematical functions rather than procedures or sub-routines.

A New Program consists of a set of equations of the form (1). For example, the program corresponding to the data processing network of figure 1 is given by:

$$\begin{array}{lcl} (S_3, S_4, S_5) & = & f_1(S_1, S_2) \\ (S_6, S_7) & = & f_2(S_3, S_4, S_5) \\ S_8 & = & f_3(S_6, S_9) \\ S_9 & = & f_4(S_4). \end{array} \tag{2}$$

In this program, variables S_3 and S_5 are never referred to separately; so we can simplify things by combining these into a new variable S_{10} to obtain [9]:

$$\begin{array}{lcl} (S_{10}, S_4) & = & f_1'(S_1, S_2) \\ (S_6, S_7) & = & f_2'(S_4, S_{10}) \\ S_8 & = & f_3(S_6, S_9) \\ S_9 & = & f_4(S_4). \end{array} \tag{3}$$

It turns out to be more convenient and more useful to re-express such equations in terms of single dependent variables. This corresponds to a preference for constructing networks whose processor nodes have out-degrees of unity. If we regard multiple output processors as *multi-purpose*, this implies a preference for designs based on *single purpose* processors. The equations then become:

$$\begin{array}{lcl} S_{10} & = & f_5(S_1, S_2) \\ S_4 & = & f_6(S_1, S_2) \\ S_6 & = & f_7(S_4, S_{10}) \\ S_7 & = & f_8(S_4, S_{10}) \\ S_8 & = & f_3(S_6, S_9) \\ S_9 & = & f_4(S_4); \end{array} \tag{4}$$

so that the original functions f_1 and f_2 have been subdivided. Since every New Program is the description of a data processing network, equations (2), (3) and (4), though representing equivalent programs (they all produce the same result), correspond to different networks. The network corresponding to New Program (4) is given in figure 2.

Only one rule must be followed in writing a New Program:

[8] i.e. x is not a location in memory.

[9] We might have regarded S_1 and S_2 in the same light save for the fact that the specification of the system will have involved S_1, S_2, S_7 and S_8

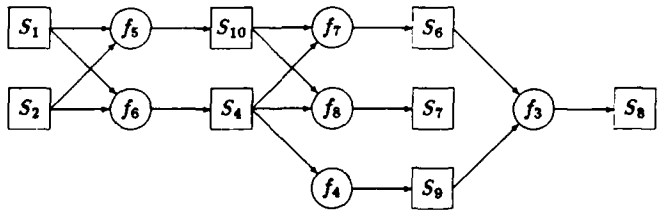

Figure 2: Network performing a New Program equivalent to the program performed by the network of figure 1.

- *A dependent variable cannot depend on more than one function (unless there are redundant equations).*

Clearly a variable cannot have different values simultaneously. In network terms, this corresponds to the rule, mentioned in section 3, that the in-degree of every store must be unity. The rule also corresponds to the "single assignment" rule first enunciated by Tesler and Enea in connection with parallelism [10]. It follows that special attention must be given to the introduction of data flow loops.

The language of a New Program has the advantage of containing only one statement and this is similar to the single *FORM* statement of MASCOT [6] [11] whose purpose is to assemble a network by creating directed arcs. For the present we shall restrict ourselves to acyclic networks, thus avoiding implicit equations like $y = f(y)$. Loops can arise in connection with implementation and minimising storage, an example being given in section 7.6.

In a New Program we need to know the "meanings" of rectangles and circles but never their actual compositions. The data structures of variables, though "understood" and defined functionally, are therefore irrelevant. Similarly, regarding processors as black boxes, the algorithm used by a process is unknown and irrelevant; it is just assumed correct and "given". In old programming, certain *basic* structures like the arrangements of bits in simple variables are irrelevant, and certain basic functions such as *add, subtract, assign* and so on are unexplained, "given" and assumed correct [12]. In New Programming all variables and functions are basic in the sense of being given, defined to be correct and unexplained. Of course, all variables have meanings given by their names [13] and all basic functions are completely understood in terms of their specified effects. Moreover, in New Programming all variables have equal status irrespective of the quantity of information appertaining. For example, a variable named "boolean" would contain one bit of information but another variable named "UK national census" would contain a vast number of bits. Both would be represented in a network diagram by single rectangles and regarded as equally important. On the other hand, the concept of *variable* variables, of fundamental importance to old programming, does not exist.

Now even though each function is defined to be correct, so that any processor looked

[10] See also [5].
[11] In MASCOT 3, however, *FORM* has been superseded by the *SUBSYSTEM* concept.
[12] Occasionally, this assumption is flawed.
[13] ie TYPES

at separately will produce a correct answer, a process can be *incorrect* or *unsafe* if extra unknown inputs or outputs exist [14]; for these may become connected with other incorrect or unsafe processes when a network is formed. It follows that we cannot prove a network correct without first establishing the non-existence of additional input and output paths to and from each process. This is an issue whose importance is often unrecognised and arises when planning the implementation of networks by old programming techniques. Although the avoidance of these unwanted paths is really quite easy once their danger is realised, some authors suggest it would be better to invent a new language to include a new structured statement: examples include *FORM* in MASCOT, already mentioned, and *PROCESS* in MODULA.

In asynchronous processing where invalid definitions temporarily exist, a process that is otherwise correct may nevertheless be unsafe if it does not terminate [15] for invalid inputs. This is also easy to solve once the problem is recognised, bearing in mind the essential stability of asynchronous systems based on acyclic networks. If all processes terminate irrespective of the validity of their inputs [16], we have a powerful method of testing systems by running the individual old programs in the "wrong order".

6 PROVING NEW PROGRAMS CORRECT

Assuming that the individual processors behave correctly, the correctness of a system as a whole is easily proved either algebraically or diagrammatically. In New Program (3), substituting for S_9 yields a *reduced* set of equations:

$$\begin{array}{lcl} (S_{10}, S_4) & = & f_1'(S_1, S_2) \\ (S_6, S_7) & = & f_2'(S_4, S_{10}) \\ S_8 & = & f_3(S_6, f_4(S_4)) = f_9(S_6, S_4), \end{array} \tag{5}$$

where f_9 is a new function combining f_3 and f_4. In network terms this is equivalent to creating a new process by drawing a continuous, "circular" curve round the functions to be eliminated. Any variable not connected with functions outside this region becomes an internal variable (eg S_9). We are no longer interested in the contents of this new region, that is to say the composition of f_9. Obviously we can substitute in this way as many times as we like and in any order, removing the eliminated equations each time until no further eliminations are possible. The New Program is correct if:

1. only one equation is left;
2. it expresses the required input-output relations of our original system exactly;
3. the inputs and outputs of each processor implemented correspond exactly to the equation (no more and no less).

In the example, we reach an overall description for the system given by

$$(S_7, S_8) = f_{10}(S_1, S_2), \tag{6}$$

[14] This refers to the semantics of scoping in some languages.

[15] ie complete its cycle

[16] Thus each function must be "total". eg *divides* must deal safely with a transient divisor that vanishes.

where S_7, S_8 are the external outputs, S_1, S_2 are the external inputs a. d f_{10} is the function of the whole system. In network terms, the progressive elimination of equations is described as *reducing* the network. Note that equation (6) also represents our starting point prior to developing New Programs like (2), (3), (4) and (5). Equation (6) is a reduction of equation (2) while (2) is an expansion of (6).

It follows that confidence in the correctness of a system developed using the techniques of New Programming has little to do with its size or complexity but rests on our confidence in the correctness of the "basic" functions.

When designing a system we should therefore try to expand it into as many, simple, functional equations as possible. This expansion, which can be performed both by writing equations and by drawing a network, is the main thrust of New Programming.

7 EXAMPLES

At this point, a page was missing from the original draft typescript which started mid-way through section 7.2. This must have included the title of a new section on examples and some narrative introducing each example. I have added the bare minimum of text for the sake of clarity.

7.1 FIVE PROBLEMS

To illustrate the essence of New Programming consider the following five examples:

1. Design a program to find the product of a pair of integers of known word-length.
2. Design a program to find the complex roots of the quadratic equation

$$az^2 + bz + c = 0,$$

   whose coefficients are complex numbers with a non-zero. Ignore issues of overflow.

3. Given a database containing the results of a national census, design a program to find the average age of people whose surnames begin with a particular letter of the alphabet.
4. Design
   - a compiler for the programming language CORAL 66;
   - a translator from CORAL 66 to a form suitable for control flow analysis.
5. Design a chess playing program.

7.2 TOP LEVEL DESCRIPTIONS

The top level descriptions of each problem, analogous to equation (6), are now presented. They are devoid of semantic detail, only the information flow relations being specified, but they form the starting points for their subsequent decompositions into New Programs. Of course, this is the reverse of the reduction process described in section 6.

1. Product of integers:

   Product of two integers of given word length =
   f(Integer1, Integer2, Number of bits).

2. Quadratic equations:

   Roots of quadratic equation = $f(a, b, c)$.[17]

3. Average age:

   Average age of people whose surnames begin with a particular character =
   f(National census, Character).

4. CORAL tools:

   (Relocatable binary code, Error messages, Nodal format for program analysis) =
   f(A CORAL 66 segment, Parameters for size).

5. Chess player:

   Black or white's next move = f(Present state of chess board, Black or white).

7.3 DECOMPOSITION

The next step for each example is to decompose the overall data transformation function into a set of sub-functions. This is achieved intuitively by selecting one or more intermediate variables that will help us find the outputs from the inputs. We can think forwards from the inputs or backwards from the outputs.

The simplest case produces a serial data flow system. Suppose we start with

$$y = f(x)$$

and can think of only one intermediate variable p, say. We can then write:

$$\begin{aligned} y &= f_1(p), \\ p &= f_2(x) \end{aligned}$$

where f_1 and f_2 are *less complex* than f. Of course, there is no need for f_1 and f_2 to be of equal complexity, the choice of p being somewhat subjective.

We may now regard each sub-equation in isolation as a separate data transforming system to be constructed. For example, if $p = f_2(x)$ were divided into two equations using another intermediate variable q, we would write:

$$\begin{aligned} y &= f_1(p), \\ p &= f_3(q), \\ q &= f_4(x). \end{aligned}$$

[17] If we were worried about a being small, we might instead start with (Roots, Error messages) = $f(a, b, c)$.

Similarly, we could continue dividing as many times as we wished until all functions were judged to be elementary. Of course, the meaning of "elementary" is somewhat arbitrary, depending on the nature of implementation. We can see this with the first example: from the programming viewpoint the function is elementary but the function would be subdivided if we wished to design a hardware multiplier using, say, half adders.

Now there is no need to restrict ourselves to purely serial networks and indeed it is unlikely that we can do so in practice. Given sufficient design experience and insight into previous methods, an initial decomposition into a network should be easily achievable. Most people lacking specialist knowledge will have immediate difficulties with example 5 and, to a lesser extent, with example 4. It follows that, beyond a certain point, decomposition may not be obvious even though *further intermediate variables must exist.* A useful technique in this event is to create multiple output functions by subdividing variables into parts [18]. For example, suppose that the variable q, introduced above, consisted of two parts, q_1 and q_2. We could write

$$(q_1, q_2) = f_8(x).$$

Is p now a function of both q_1 and q_2? We may be able to envisage a further variable r necessary for computing both q_1 and q_2 such that:

$$\begin{aligned} r &= f_5(x), \\ q_1 &= f_6(r), \\ q_2 &= f_7(r) \end{aligned}$$

When invention finally dries up we are forced to write the individual functions as separate, old programs. If these turn out to be trivial as measured by the time taken to write them, then no further subdivision is required: otherwise it should rapidly become obvious how to subdivide further. The temptation to persist with old programming beyond this point should be resisted since over-optimism afflicts even experienced programmers!

7.4 QUADRATIC EQUATIONS

To illustrate the technique of designing and implementing New Programs let us decompose a simple example where the design difficulties described in section 7.3 do not arise. In the second example it is clear that we have two roots and that a suitable intermediate variable is $p = \sqrt{b^2 - 4ac}$. Thus we may write (ignoring distinctions between functions [19]):

$$\begin{aligned} roots &= f(root_1, root_2), \\ root_1 &= f(a, b, p), \\ root_2 &= f(a, b, p), \\ p &= f(a, b, c). \end{aligned} \tag{7}$$

Introducing another intermediate variable q, namely the square of p, this becomes:

$$\begin{array}{lcll} roots & = & f(root_1, root_2), & \text{ie transfer to output} \\ root_1 & = & f(a, b, p), & \text{ie } root_1 = (-b + p)/2a \\ root_2 & = & f(a, b, p), & \text{ie } root_2 = (-b - p)/2a \\ p & = & f(q), & \text{ie } p = \sqrt{q} \\ q & = & f(a, b, c). & \text{ie } q = b^2 - 4ac \end{array} \tag{8}$$

[18] This seems close to the spirit of data refinement.

[19] ie specifying the information flow relations only

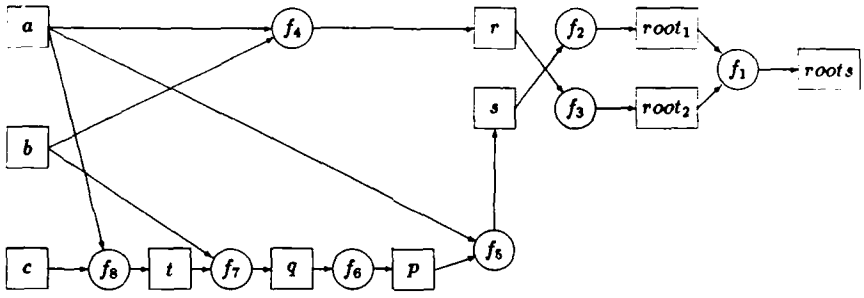

Figure 3: Data Processing Network for solving quadratic equations.

A further subdivision may be achieved by calculating $r = b/2a$, $s = p/2a$ and $t = 4ac$ so that we can write formally (assuming operations on complex numbers to be "basic"):

$$\begin{array}{lcll} roots & = & f_1(root_1, root_2), & \text{ie transfer to output} \\ root_1 & = & f_2(r,s), & \text{ie } root_1 = (-r+s) \\ root_2 & = & f_3(r,s), & \text{ie } root_2 = (-r-s) \\ r & = & f_4(a,b), & \text{ie } r = b/2a \\ s & = & f_5(a,p), & \text{ie } s = p/2a \\ p & = & f_6(q), & \text{ie } p = \sqrt{q} \\ q & = & f_7(b,t), & \text{ie } q = b^2 - t \\ t & = & f_8(a,c). & \text{ie } t = 4ac \end{array} \tag{9}$$

Note finally that progressive substitution proves this New Program correct. The corresponding network is presented in figure 3.

7.5 OFF-LINE IMPLEMENTATION

Let us suppose that New Program (9) is intended for an off-line system. Thus, a single set of parameters a, b, c is input and a single pair of roots is required. We can naturally assume a synchronous system where the functions are to be performed in series using a single computer. We can write down a possible sequence by starting with the input and performing each function as soon as all its inputs are defined. (Note, however, that *any* asynchronous sequence would do as long as it were continuous.)

The network illustrated in figure 3 shows that, for a discontinuous, data flow system employing synchronous, serial processing, any sequence takes the form f_8, f_7, f_6, f_5, with f_4 performed at any stage, followed by f_2 and f_3 in arbitrary order, and then f_1. A possible sequence is therefore f_8, f_7, f_6, f_5, f_4, f_3, f_2, f_1, and a conventional program could be

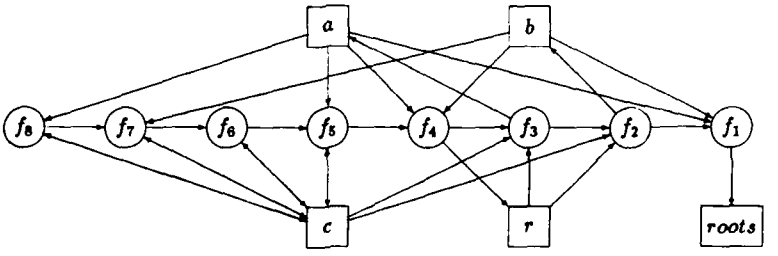

Figure 4: Data Processing Network for solving quadratic equations with reduced storage. Directed arcs joining processors indicate control structure.

written in assignment form as:

$$
\begin{aligned}
&t := f8(a, c);\\
&q := f7(b, t);\\
&p := f6(q);\\
&s := f5(a, p);\\
&r := f4(a, b);\\
&root2 := f3(r, s);\\
&root1 := f2(r, s);\\
&roots := f1(root1, root2).
\end{aligned}
\tag{10}
$$

7.6 MINIMISING STORAGE

Since store c is immediately free, we may use it as a composite variable to include variables t, q, p and s. At a later stage we may use a and b to store the two roots. This yields a program with reduced storage:

$$
\begin{aligned}
&c := f8(a, c);\\
&c := f7(b, c);\\
&c := f6(c);\\
&c := f5(a, c);\\
&r := f4(a, b);\\
&a := f3(r, c);\\
&b := f2(r, c);\\
&roots := f1(a, b).
\end{aligned}
\tag{11}
$$

Figure 4 indicates the complexity of the network equivalent of program (11). Two types of directed arcs are now needed. Those joining processors indicate control structure. Note also that sharing storage has created data flow loops for example from c to f_8 and back [20]. The general problem of optimal storage sharing is discussed by Marill.

If lists of values were to be input then the functions would be implemented with *FOR LOOPS* or their equivalent. If the lists were very long the stores could be held on discs, for example, and the programs would be those of an operating system. Given that the

[20] though these do not really correspond to equations like $y = f(y)$

elemental processes are correct then program (10) is a correct program. Also, if we can show that program (11) is equivalent to program (10), then it too is correct on the same assumptions. In practice, one should develop and test New Programs like (10) before implementing the storage sharing versions like (11).

If we were implementing a parallel processing system, figure 3 and its equivalent New Program indicate that f_4 and f_8 and later on f_2 and f_3 may be performed in parallel but that all the other functions must be performed in series. In general, parallel processing would be expected to reduce the opportunities for sharing storage though not in this particular example.

8 OLD PROGRAMS VIEWED AS DATA PROCESSING NETWORKS

An old program may be regarded as a data processing system of stores and processors that uses *synchronous serial processing*. However, the important distinction between the data flow network and the external controller, previously discussed, is abolished. It is the essence of old programming that the controlling "sequencer" triggers processors in an order that is no longer completely pre-arranged but where successor processors are sometimes *selected* [21]. To do this, the sequencer "reads" certain (boolean) values of a predetermined subset of the stores. Consequently, whether processors will be triggered will depend on the initial values placed in the external input stores. The network thereby becomes a static description of all possible data stores, data flow paths and processes from which one particular path and set of processes is chosen for a given set of external input values. A new distinction thus arises between static and dynamic descriptions of the network that does not exist in New Programming.

In system terms, an old program is therefore more difficult to analyse and understand than its new counterpart since both data flow and control sequence must be considered simultaneously. The modified network diagram of even a trivial old program becomes incomprehensible in human terms and is useful only as a model for automatic analysis. In fact the complexities of control flow are such that any useful diagram (eg a flow diagram) concerns this aspect alone and a program text in which the human reader performs the role of sequencer is the only feasible description. In practice, system complexity and the importance of control "structure" are greatly increased by introducing *variable* stores (computed addresses, indexed arrays) and *variable* processors (subroutines, procedures). Complexity is also increased by abolishing the distinction, on which we have insisted, between design correctness and efficiency. Sharing storage, for example, is achieved by using *program* variables rather than mathematical variables and by using procedures and subroutines so that processors are "called" to perform different but similar processes at different times. (Procedures with parameters are equivalent to processors with variable input-output stores, the connections being made at the time of "calling".)

It follows that the skill of the conventional programmer is concerned essentially with constructing a particular solution to a problem from a sequence of processors using stores whose composite names refer to abstract structures of store *locations*. Much of his time is spent in checking the sequence of operations that ensue when a particular set of data

[21] eg by means of *IF..THEN..ELSE..* constructs

is input to the system, since there is no guarantee of proving the correctness of such a system. In New Programming, no programming skill is required but a deep knowledge of the application problem and of general methods of solution is essential. With old programming, the reverse is true: apart from a knowledge of programming, all that is required is the invention of a particular sequential algorithm to solve the problem. Since inventing *ad hoc* sequential algorithms (of varying quality) is easy for almost any problem, it follows that the writing of old programs, like any form of unplanned writing, can be a virtue or a vice. When used as a trial and error method of *learning* about possible solutions and thereby about the subject of the particular application, it is a virtue. But when it becomes a means of *evading* serious study of the application and of possible better methods of solution, it is a vice. It must be admitted that, as a pastime, problem solving is far more attractive than design! This is a difficulty faced by advocates of previous design approaches like "structured programming".

Since the only possible obstacle to New Programming is an inability to subdivide a complex function or variable, for design is not always easy, we can use old programming as a highly effective last resort, namely as a learning tool. Of course, old programming must be pursued no further than necessary to achieve the particular subdivision. It might be thought from all this that New Programs like old programs evolve through continual modification but experience so far suggests the contrary.

9 TRANSLATING OLD INTO NEW PROGRAMS

It is possible to translate an old program mechanically into non-sequential form and this is frequently discussed in the literature for the purposes of parallel programming. A transformation into data flow form follows a similar course but the objectives are somewhat different. The full scale transformation is rather complex and involves:

1. the separate naming of composite named variables,
2. the determination of the complete conditions for each assignment, and
3. the expression of each dependent variable in the form

$$y = f(a, b, ...p_1, p_2, ...), \tag{12}$$

where $a, b, ...$ are the independent variables and $p_1, p_2, ...$ are the predicate variables.

Since we are not concerned in New Programming with abstract structures, all operations on such structures can be grouped together. For example, given an array *invert*, *invert*[3] has meaning in New Programming but *invert*[*i*], with *i* an integer variable, has no meaning since we cannot permit variable stores.

In New Programming there is no objection to naming a group of stores but the group's name is superfluous to the program. On the other hand, to require this group to have some particular abstract structure, like "array", is foreign to New Programming. We must therefore regard the names of old programming objects as basic elements whose structure is irrelevant. For example, in the case of arrays or simple identifiers that can be addressed as part-words [22], there is the option either to regard them as basic stores or to analyse

[22] a CORAL construct

the program in greater detail so that each array element or bit/byte is basic. The latter option is almost certainly too detailed in practice unless we wish to design a machine code or hardware analogue of the program.

An alternative to the automatic transformation of an obese, old program is manual division. Here we view the program from the "top", or outer block, starting from its first operation. This outer block must be converted into a sequence without selective modules (*IF...THEN...ELSE...* statements) in which no path loops back to a previous module.

There are two types of loops in old programming, one for which the number of iterations is predetermined (eg the *FOR* loop) and the other where it depends on the current state of data (eg the *WHILE...DO* loop). The former can be regarded as part of New Programming while the latter lies within the letter if not the spirit of New Programming. Similarly, we can permit loops within loops *ad infinitum*.

The problem, therefore, is to distinguish between iterative and selective modules and to determine the input and output data of each module. Where a program is written with the outer block consisting of procedure calls, an easy method of subdivision is to transform each procedure into a process, *regarding each call as a separate process.* The most difficult part of any transformation is the determination of the input and output data. With Algol-based languages such data must have been declared in the outer block. Unfortunately, declarations in the outer block may also include data that is private to each procedure, that is to say whose name is shared (composite names). The parameter list on the other hand usually includes some (but not necessarily all) of the input-output data.

A selective module must be removed from the controlling sequence by converting it into a process into which the predicate is "absorbed". In Algol-based languages, the simplest solution is to convert the selective module into a procedure (called once). If the parameter mechanism is adequate the input and output data could be passed solely via parameters, but it is probably safer to use procedures without parameters. (This is essential if the input-output data is to reside on backing store.)

Any mechanical transformation is crude since it ignores the purpose of the program, so that the methods just described should be regarded only as a guide. In many cases, a program is too large because, in the name of efficiency, it is attempting to perform many different functions and to minimise storage at the same time. Conventional programmers find that transforming their programs, by decomposing them into data transforming functions, is an acceptable technique since it clarifies what they admit is something of a muddle. However, a design concept that provides a new store for each definition instead of "updating" an old store is regarded as tedious and is therefore avoided particularly when composite store names are to be re-introduced at the implementation stage.

Nevertheless, the principle of separating correctness of design from store sharing is important for building reliable systems from distinct "processors" that are easily tested, understood and re-usable by *other* programmers. Moreover, if composite names are introduced at the design stage, an opportunity for meaningful distinction between the separate names is lost. In addition, system testing is less satisfactory since the overall result depends on the sequential order in which the separate procedures are called.

10 ORGANISATIONAL BENEFITS

From a manager's viewpoint, and even from a programming manager's viewpoint, a data flow network represents an improvement in "modular design" over any possible, structured, old program; for in the latter the various sub-functions must *co-operate* in the sense that the *correctness* of the overall system is highly dependent on the order in which they are performed. In data flow design, the separate functions do *not* co-operate and so the tasks of writing the trivial, old programs are unrelated. The manage can control software production easily without any knowledge of programming whatever. For example, he can follow progress simply by asking how many processors are working correctly and how many remain unfinished, whether any are delayed and if so who is writing them, in the secure knowledge that, if the system as a whole does not work, the fault cannot possibly lie with old programs but must lie with the system designer.

On the other hand, the programmer must merely demonstrate that his individual program produces valid outputs for valid inputs: his job becomes more clearly defined than if his program were part of a larger program. Since his job has been simplified, and even trivialised, a snag might be that he finds it less interesting. In fact certain very small programs might become standardised and designed solely for efficiency.

It is true that a heavy responsibility lies with the data flow designer when the time comes to integrate the "processors" and "data areas" into a working system, for he alone is accountable for any problems. Fortunately, experience in the real-time field shows that data flow designs are easy to develop, change little during development and offer few integration problems even when the network functions are split between different firms. The integration problems that remain are usually not network design faults but concern *rates of data flow* through the network.

11 EDITOR'S POSTSCRIPT

I am most grateful to several colleagues for having devoted much time and effort to a critical reading of this paper. S C Giess, T L Thorp, T A D White and J M Whitfield have all made suggestions that have added clarity to the text while rectifying a few misconceptions on my part.

As regards the paper itself it is clear that, although much was novel when first envisaged in the late seventies, the subject itself has moved on a great deal. (For a modern reference JMW has suggested [7].) For my own part, the preparation of this script has been an education. I conclude with some comments that relate the topic of this paper to some recent developments concerning the correctness of data flow networks without loops.

Consider, for example, the network depicted in figure 2 with the S_i representing data stores that are written destructively and read non-destructively. Let us regard the figure as a design for a system that may be implemented in four different ways:

(SYNCHRONOUS, ASYNCHRONOUS) × (SERIAL, PARALLEL).

Three of these may be modelled straightforwardly in a form suitable for static analysis and verification. Possibilities include MALPAS IL [8] and SPADE FDL [9]. In IL, for example, the declarative part of the model takes the following form in each case:

```
TYPE store1, store2, ..... , store10;  [Need not all be different]
FUNCTION f5(store1, store2): store10;  [Similarly for the other f s]

PROCSPEC process5(IN x: store1         [Similarly for other processes]
                  IN y: store2
                  OUT z: store10)
DERIVES z AS f5(x, y)
POST    z = f5('x, 'y);                [' denotes initial state]

PROCSPEC network(IN s1: store1
                 IN s2: store2
                 OUT s7: store7
                 OUT s8: store8)
POST input_output_relations('s1, 's2, s7, s8);
```

Of course the body of *network* will depend on choice of implementation. For synchronous, serial processing, a possible body would be:

```
PROC network;        [SYNCHRONOUS SERIAL]
VAR s10: store10;  VAR s4: store4;
VAR s6: store6;    VAR s9: store9;
  process5(s1, s2, s10);
  process6(s1, s2, s4);
  process7(s10, s4, s6);
  process8(s10, s4, s7);
  process4(s4, s9);
  process3(s6, s9, s8)
ENDPROC
```

and this should be capable of proving correct or otherwise.

For synchronous, parallel processing, the *MAP....ENDMAP* construct provides a model of parallelism and a possible body would be:

```
PROC network;        [SYNCHRONOUS PARALLEL]
VAR s10: store10;  VAR s4: store4;
VAR s6: store6;    VAR s9: store9;
  MAP
    process5(s1, s2, s10);
    process6(s1, s2, s4)
  ENDMAP;
  MAP
    process7(s10, s4, s6);
    process8(s10, s4, s7);
    process4(s4, s9)
  ENDMAP;
  process3(s6, s9, s8)
ENDPROC
```

Again, verification presents no serious problem.

For asynchronous, serial processing we need to introduce a loop and one of many possible bodies is given by:

```
PROC network;        [ASYNCHRONOUS SERIAL]
VAR s10: store10;  VAR s4: store4;
VAR s6: store6;    VAR s9: store9;
VAR counter: integer;
  counter := 0;
  LOOP
    ASSERT loop_invariant;
    EXIT WHEN counter > 10;
    process3(s6, s9, s8);
    process4(s4, s9);
    process5(s1, s2, s10);
    process8(s10, s4, s7);
    process7(s10, s4, s6);
    process6(s1, s2, s4);
    counter := counter + 1
  ENDLOOP
ENDPROC
```

However, in implementations of this nature the loop invariant needs to carry the state of the system on each cycle of the computation. Consequently, the resources required in time and space to complete the verification process can be prohibitive.

Finally, there is no obvious method of modelling asynchronous, parallel processing. No matter: for we know that a proof of one amounts to a proof of all. In other words it suffices, at least for a one shot system, to prove the correctness of the design of figure 2 by proving the correctness of one of the synchronous implementations. As regards asynchronous implementations, the proof of correctness still holds with the caveat that each process completes its cycle even for illegal inputs. Of course we must take care to say that the input-output relations are satisfied *eventually*, that is after the transient period of invalid definitions.

With regard to the processing of continuous streams of data by asynchronous, parallel means, there remains the issue of maximum allowable flow rates. (If a thermometer is required to measure the temperatures of fluids in several glass jars, the correct results will *never* be obtained by allowing it only five seconds in each jar instead of the recommended sixty. In fact such analogies with analogue systems motivated Bob Phillips throughout his work.) The maximum flow rate for a given network will depend on the execution time (period) of each process, the communication times between processors and the topology of the network. A detailed study of this, however, lies outside the scope of this paper.

References

[1] C S E Phillips
Networks for real-time programming
Computer Journal, **10** 1, pp 46 - 52, 1967.

[2] K Jackson and H R Simpson
MASCOT - A modular approach to software construction operation and test
RRE Tech Note 778, 1975.

[3] C S E Phillips
Structural analysis of programs for automatic maintenance and other purposes
Article in "Pragmatic programming and sensible software": Online 1978.

[4] H R Simpson and K Jackson
Process synchronisation in MASCOT
Computer Journal, **22** 4, pp 332-345, 1978.

[5] D D Chamberlin
The single assignment approach to parallel processing
AFIPS Fall Joint Computer Conference, **39**, pp 263-269, 1971.

[6] The official handbook of MASCOT 2: JIMCOM 1983.

[7] J A Sharp
Data flow computing
Ellis Horwood Ltd, 1985.

[8] Rex, Thompson and Partners
MALPAS Intermediate Language.

[9] Program Validation Ltd
SPADE FDL Manual.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... Unclassified .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memorandum 4249		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	Royal Signals and Radar Establishment St Andrews Road, Great Malvern Worcestershire WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

The New Programming

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Phillips C S E	Bramson B D		1989.5	23

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The late C S E Phillips's unfinished paper is presented. The sequential mode of thinking that underpins conventional programming is cited as being unnecessary, over-complicated and impractical for all but the most trivial of problems. In New Programming, the flow of data is the only factor of importance. A New Computer is a non-sequential machine whose elemental processors are not required to execute their processes in an ordered sequence. Such a machine is easier to design and easier to prove correct than its sequential counterpart.